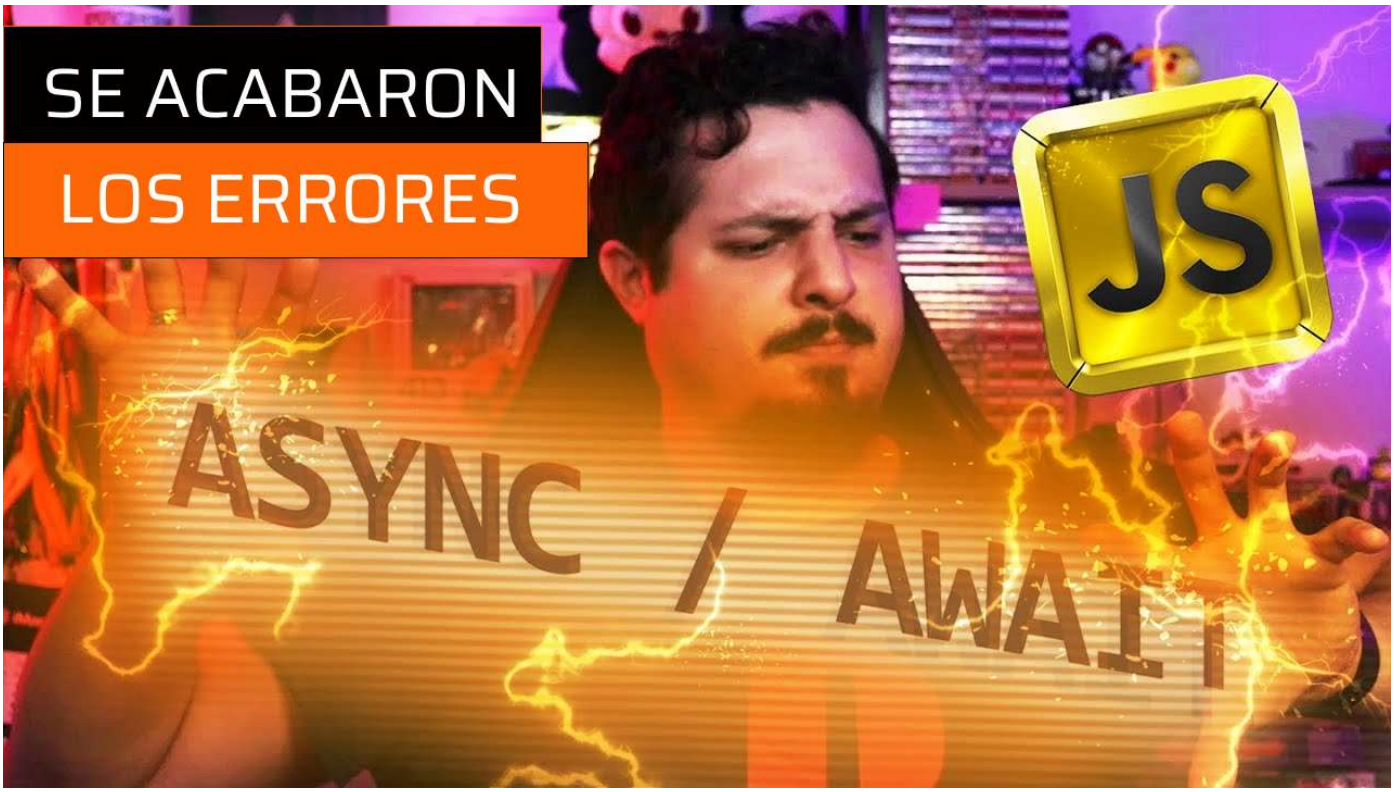
Async/await en JavaScript: ¿qué es y cuándo usar programación asíncrona?

J

juliana-amoasei

28/04/2022



En el dia a dia del desarrollo web, usamos muchos (y cada vez más) datos externos, por ejemplo, datos recibidos a través de un *endpoint* de una API REST o los resultados de algún otro procesamiento. Es decir, cuando esto ocurre, el sistema tiene que esperar a que "lleguen" los datos o que ocurra algún evento antes de utilizar estos datos. Lo llamamos *programación asíncrona* cuando nuestro código debe esperar a que ocurra cierto

procesamiento que no está bajo nuestro control, como una requisición externa, y luego continuar con la ejecución de la siguiente tarea.

Trabajando con front-end, vemos que una buena parte de lo que sucede a nivel del navegador está dirigido por eventos o *event-driven*. Es decir, el código espera a que suceda algún evento (por ejemplo, el usuario hace clic en un botón) antes de ejecutar cualquier código. Otros ejemplos de eventos, además de los clics del mouse, son tocar la pantalla, presionar una determinada tecla, pasar el cursor del mouse sobre un elemento, etc. Pero, además de estas interacciones del usuario con la interfaz, existen muchas otras situaciones que pueden ser síncronas o asíncronas.

Por ejemplo, podemos pensar en la comunicación. Una llamada telefónica es un ejemplo de comunicación síncrona: cuando hablamos por teléfono, la información entra y sale en secuencia, una tras otra; hacemos una pregunta, inmediatamente recibimos la respuesta, con los datos de esa respuesta hacemos otro comentario, etc.

Por otro lado, una conversación online a través de algún messenger, como WhatsApp o Telegram, es un ejemplo de comunicación asíncrona: enviamos un mensaje y no nos quedamos mirando la pantalla, esperando, hasta que la otra persona responda (¡o al menos no deberíamos!). Después de todo, no tenemos forma de saber cuándo y si llegará esa respuesta. Enviamos el mensaje y vamos a hacer otras cosas mientras no llega la respuesta, a diferencia del teléfono.

Con el código, el proceso es similar: un código síncrono es uno que ocurre en secuencia, una instrucción tras otra.

```
function soma(num1, num2) {
 return num1 + num2;
}

console.log(soma(2, 2)) // 4
```

Hasta aquí, todo esta bien! JavaScript ejecutó una línea tras otra.

Pero, ¿qué sucede cuando, por ejemplo, nuestro código necesita recibir algunos datos de una API? Si bien es necesario esperar la solicitud y respuesta de la API, no podemos bloquear el funcionamiento de todo nuestro programa; sería lo mismo que enviar un mensaje por WhatsApp y esperar la respuesta sin hacer nada más mientras tanto.

Es para este tipo de situaciones, que se requiere un **procesamiento asíncrono**, que existen las *Promises*. El significado de *Promise* en JavaScript es similar al literal: Una persona te da el contacto de Telegram y te pide que le envíes un mensaje, prometiendo que le responderás... Cosa que no tenemos forma de saber si sucederá.

Cuando enviamos una *requisición* de datos a una API, tenemos la promesa de que estos datos llegarán, pero hasta que eso suceda, el sistema debe seguir funcionando. Si, por ejemplo, el servidor no funciona, esa promesa de datos podría no cumplirse y tenemos que lidiar con eso. Las promesas funcionan en este contexto.

Existen dos formas de trabajar con el procesamiento asíncrono (es decir, *Promises*) en JavaScript: usando el método `.then()` o las palabras clave `async` y `await`.

Usando Promesas con .then()

Ya que hablamos de las API's REST, veamos un ejemplo usando la Fetch API de JavaScript para obtener datos y convertirlos al formato JSON. Esta API (que funciona de forma nativa en los navegadores actuales) tiene algunos métodos internos y, de forma predeterminada, devuelve una *Promise* que resolverá la requisición, tenga éxito o no.

```
function getUser(userId) {
 const userData = fetch(`https://api.com/api/user/${userId}`)
   .then(response => response.json())
   .then(data => console.log(data.name))
}

getUser(1); // "Nombre Apellido"
```

Desglosando el código anterior: la función `getUser()` recibe un id de usuario como parámetro, de modo que sea pasado el *endpoint* REST ficticio. El método `fetch()` recibe el *endpoint* como parámetro y devuelve una *Promise*.

¿Y cómo funcionan las Promesas? Las promesas tienen un método llamado `.then()`, que recibe una función *callback* y retorna un "objeto-promesa". **No es un retorno de los datos, es la *promesa* del retorno de esos datos.**

Entonces, podemos escribir el código de lo que sucederá a continuación con los datos recibidos por *Promise*, y JavaScript esperará a que *Promise* se resuelva sin pausar el flujo

del programa.

El resultado puede o no estar listo todavía, y no hay forma de obtener el valor de una Promesa de forma síncrona; Solo es posible pedirle a Promise que ejecute una función *callback* cuando el resultado esté disponible, ya sea lo que se solicitó (los datos de la API, por ejemplo) o un mensaje de error si algo salió mal con la solicitud (el servidor puede estar fuera de servicio, por ejemplo).

En el ejemplo anterior: cuando iniciamos una cadena de promesas, por ejemplo para hacer una solicitud HTTP, mientras la respuesta está pendiente, retorna un `Promise object`. El objeto, a su vez, define una instancia del método `.then()`. En lugar de pasar la función *callback* directamente a la función inicial, se pasa a `.then()`. Cuando llega el resultado de la solicitud HTTP, el cuerpo de la solicitud se convierte a JSON y este valor convertido se pasa al siguiente método `.then()`.

La cadena de funciones `fetch() .then() .then()` no significa que se utilicen varias funciones *callback* con el mismo objeto de respuesta, sino que cada instancia de `.then()` a su vez devuelve una `new Promise()`. Toda la cadena se lee de forma síncrona en la primera ejecución y luego se ejecuta de forma asíncrona.

Captura de errores con promesas

El manejo de errores es importante en las operaciones asincrónicas. Cuando el código es síncrono, puede generar al menos una excepción incluso sin ningún tipo de manejo de errores. Sin embargo, en asincrónico, las excepciones no controladas a menudo pasan sin previo aviso y se vuelve mucho más difícil de depurar.

No hay forma de usar `try/catch` cuando se usa `.then()`, ya que el cálculo solo se realizará después de devolver el objeto-Promise. Luego debemos pasar funciones que ejecuten las alternativas, en caso de éxito o fracaso de la operación. Por ejemplo:

```
function getUser(userId) {
 const userData = fetch(`https://api.com/api/user/${userId}`)
   .then(response => response.json())
   .then(data => console.log(data.name))
   .catch(error => console.log(error))
   .finally(() => /*{ aviso de fin de cargamento }*/)
}
```

```
getUser(1); // "Nombre Apellido"
```

Además del método `.then()` que recibe el objeto-Promise para ser resuelto, el método `.catch()` retorna en caso de rechazo de la Promesa. Además, el último método, `.finally()`, se llama independientemente de si la promesa tiene éxito o falla, y la función *callback* de este método siempre se ejecuta en último lugar y se puede usar, por ejemplo, para cerrar una conexión o dar algún aviso de finalización.

Resolviendo varias promesas

En el caso de múltiples promesas que se pueden hacer en paralelo (por ejemplo, algunos datos en diferentes *endpoints* REST), se puede usar `Promise.all`:

```
const endpoints = [
 "https://api.com/api/user/1",
 "https://api.com/api/user/2",
 "https://api.com/api/user/3",
 "https://api.com/api/user/4"
]

const promises = endpoints.map(url => fetch(url).then(res => res.json()))

Promise.all(promises)
 .then(body => console.log(body.name))

Una Promesa puede estar "pendiente" (pending) o "resuelta" (settled). Los esta
```

Usando async/await

Las palabras clave `async` y `await`, implementadas a partir de ES2017, son una sintaxis que simplifica la programación asíncrona, facilitando el flujo de escritura y lectura de código; por lo que es posible escribir código que funcione de forma asíncrona, pero que se lea y

estructure de forma síncrona. `Async/await` funciona con código basado en Promises, pero oculta las promesas para que la lectura sea más fluida y sencilla de entender.

Al definir una función como `async`, podemos usar la palabra clave `await` antes de cualquier expresión que retorne una promesa. De esta forma, la ejecución de la función externa (la función `async`) se pausará hasta que se resuelva la Promesa.

La palabra clave `await` recibe una Promesa y la convierte en un valor de retorno (o genera una excepción en caso de error). Cuando usamos `await`, JavaScript esperará hasta que finalice la Promesa. Si se completa con éxito (el término utilizado es `fulfilled`), el valor obtenido es retornado. Si la Promesa es rechazada (el término utilizado es `rejected`), se retorna el error arrojado por la excepción.

Un ejemplo:

```
let response = await fetch(`https://api.com/api/user/${userId}`);
let userData = await response.json();
```

Solo puede usar `await` en funciones declaradas con la palabra-clave `async`, así que agréguela:

```
async function getUser(userId) {
 let response = await fetch(`https://api.com/api/user/${userId}`);
 let userData = await response.json();
 return userData.name; // no es necesario await en el return
}
```

Una función declarada como `async` significa que el valor de retorno de la función será, "por dentro de javascript", una Promesa. Si la promesa se resuelve normalmente, el objeto-promesa retornará el valor. Si arroja una excepción, podemos usar try/catch como estamos acostumbrados en los programas síncronos.

Para ejecutar la función `getUser()`, ya que retorna una Promesa, puedes usar `await`:

```
exibeDatosUser(await getUser(1))
```

Recordar que `await` solo funciona si está dentro de otra función `async`. Si todavía no estás familiarizado, puedes usar `.then()` normalmente:

```
getUser(1).then(exibeDadosUser).catch(reject)
```

Resolviendo varias promesas

En el caso de múltiples promesas que se pueden hacer en paralelo (por ejemplo, algunos datos en diferentes *endpoints* REST), puedes usar `async/await` junto con `Promise.all`:

```
async function getUser(userId) {
 let response = await fetch(`https://api.com/api/user/${userId}`);
 let userData = await response.json();
 return userData;
}

let [user1, user2] = await Promise.all([getUser(1), getUser(2)])
```

¿Hay diferencias entre .then() y async/await?

En términos de sintaxis, el método `.then()` posse una sintaxis con más sentido cuando se usa JavaScript de manera funcional, mientras que el flujo de declaraciones con `async/await` tiene sentido cuando se usa en métodos de clase.

`async/await` surgió como una opción de "lectura más fácil" que `.then()`, pero es importante tener en cuenta que **estos métodos no son lógicamente equivalentes**: mientras que `async/await` realiza el procesamiento secuencialmente, las promesas con `.then()` se procesan en paralelo , lo que hace que este método sea más rápido. `async/await` simplifica la escritura y la interpretación del código, pero no es tan flexible y solo funciona con una Promesa a la vez.

Cómo resolver este tipo de casos, por ejemplo, solicitar una matriz de id de pedido de un cliente en particular de una tienda:

```
async function getCustomerOrders(customerId) {
 const response = await fetch(`https://api.com/api/customer/${customerId}`)
 const customer = await response.json()

 return await Promise.all(customer.orders.map(async (orderId) => {
   const response = await fetch(`https://api.com/api/order/${orderId}`)
```

```
    const orderData = await response.json()
    return orderData.amount
  }))
}
```

En el caso anterior, usamos `Promise.all` para hacer las peticiones en paralelo, sin esperar a que vuelva la anterior para hacer la siguiente.

Iteraciones con async/await

Pero, ¿y si necesitamos manejar varias Promesas, pero no queremos hacerlo en paralelo? Un ejemplo clásico de esta situación es acceder a una base de datos con miles de registros. En este caso, no queremos que todas las solicitudes se realicen en paralelo, ya que el exceso de solicitudes simultáneas puede causar problemas de rendimiento e incluso la caída del servicio.

En este caso, `async/await` es más adecuado, ya que resolverá una Promesa a la vez.

```
async function printCustomer(customerId){
 //lógica fictícia de la función
}

async function getAndPrintAllCustomers() {
 const sql = 'SELECT id FROM customers'
 const customers = await db.query(sql, [])
 for (const customer of customers) {
   await printCustomer(customer.id)
 }
}
```

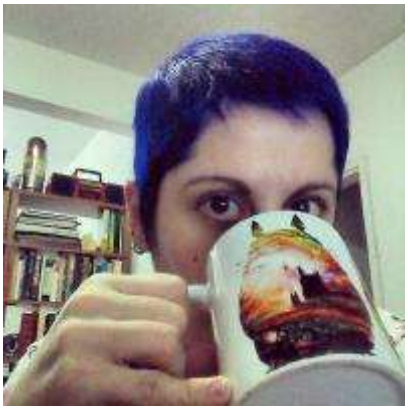
En el caso anterior, no queremos hacer todas las solicitudes a la base de datos a la vez, sino secuencialmente.

Conclusión

En este artículo, aprendimos qué son las **Promises** y cómo trabajar con programación asíncrona en JavaScript usando `.then()` y también el método más moderno `async/await`.

¿Te gustó el artículo y quieres saber más? Aquí en Alura contamos con la Formación Front End, para que aprendas y te desarrolles.

Juliana Amoasei



Programadora JavaScript con formación multidisciplinar, siempre aprendiendo a enseñar y viceversa. Trabajo en varias iniciativas de inclusión tecnológica desde 2018 y creo en el potencial del conocimiento como agente de cambio personal y social. Actualmente trabajo como instructora en la Escuela de Programación Alura y doy tutoría técnica a principiantes en desarrollo web frontend y backend.

Este articulo fue adecuado para Alura Latam por: Jose Charris

Productividad y Calidad de Vida | Transformación Ágil | Marketing Analytics | Liderazgo y Gestión de Equipos | Startups y Emprendimiento